

BY BOIAN MITOV

In the previous 2 articles, we introduced Visuino – a new graphical development environment for Arduino, and we demonstrated how you can use Delphi and the upcoming CommunicationLab from Mitov Software to communicate with Arduino, and get the data from its sensors into your applications.

It surely is a lot of fun to use Visuino and Delphi to talk to Arduino, but it would be even bigger fun to be able to create your own components for Visuino and use them in your projects or share them with other developers, the same way as you do in Delphi.

When I started developing Visuino, one of my early goals was to develop it as open and expandable platform. Since the underlying OpenWire Studio already was developed around the concept of installable components, and packages, Visuino inherited this architecture.

From the beginning, all components in Visuino were parts of installable packages, and new components could be added, by simply installing new packages similar way as new components can be installed in Delphi.

As Visuino approached its release version, I also developed and released a Beta of the Visuino Components development SDK. The SDK is available for download from the Visuino Google+ community - <https://plus.google.com/communities/116125623808250792822>.

In this article you will learn how you can develop your own components for Visuino.

START DEVELOPING VISUINO COMPONENTS

Before you start developing Visuino components, you will need to install Visuino and the Visuino components development SDK.

The SDK currently requires Delphi XE8 to be used for the component development.

The typical Visuino installation will place the executable under `C:\Program Files\Mitov\Visuino` or `C:\Program Files(x86)\Mitov\Visuino`, depending on the version of Windows you use.

The Visuino directory will contain the executable `Visuino.exe`. There is also a **Demos sub-directory**. There is another typically empty directory called **Component Packages**. This is the directory where additional component packages can be installed. In addition to this folder structure, Visuino also installs **C++** Arduino files into "Mitov" sub-directory of the Arduino library folder. Typically the Arduino libraries folder is `My Documents\Arduino\libraries`. Each Visuino component consists of 2 parts. The Arduino C++ code in the `libraries\Mitov` directory, or another `libraries*` sub-directory, and a corresponding Delphi component in a package in the **Component Packages** directory.

The C++ code is the one that compiles and executes in Arduino. The Delphi component is almost just an empty shell, that has some attributes needed to describe the capabilities of the C++ code. From that point of view, in most cases the Delphi code is just a simple description of the component. In some cases however, some more complex code generation logic can be implemented in the Delphi component, to control the way the Arduino code is generated.

The Components SDK after installation, creates its own folder structure. Currently, the typical installation will create "OpenWire" and "LabPacks" sub-folders under the Delphi main folder.

In the `LabPacks\Mitov\Arduino\Examples` folder, there is an example of component package for Visuino.

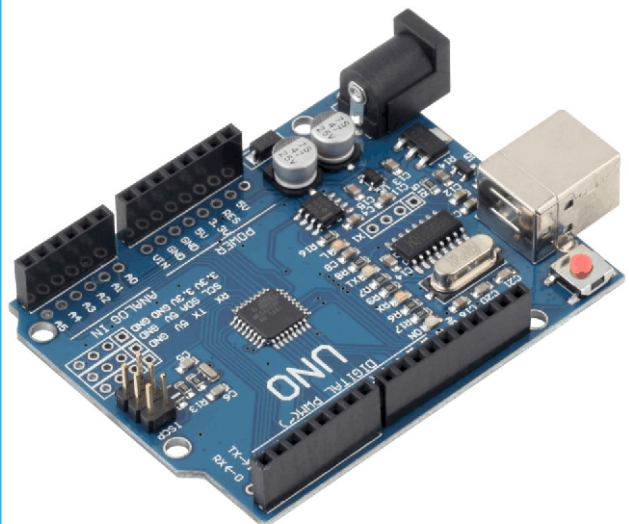
In the `C:\Program Files\LabPacks\Mitov\Arduino\Examples\library` folder is where the C++ code for Arduino is located.

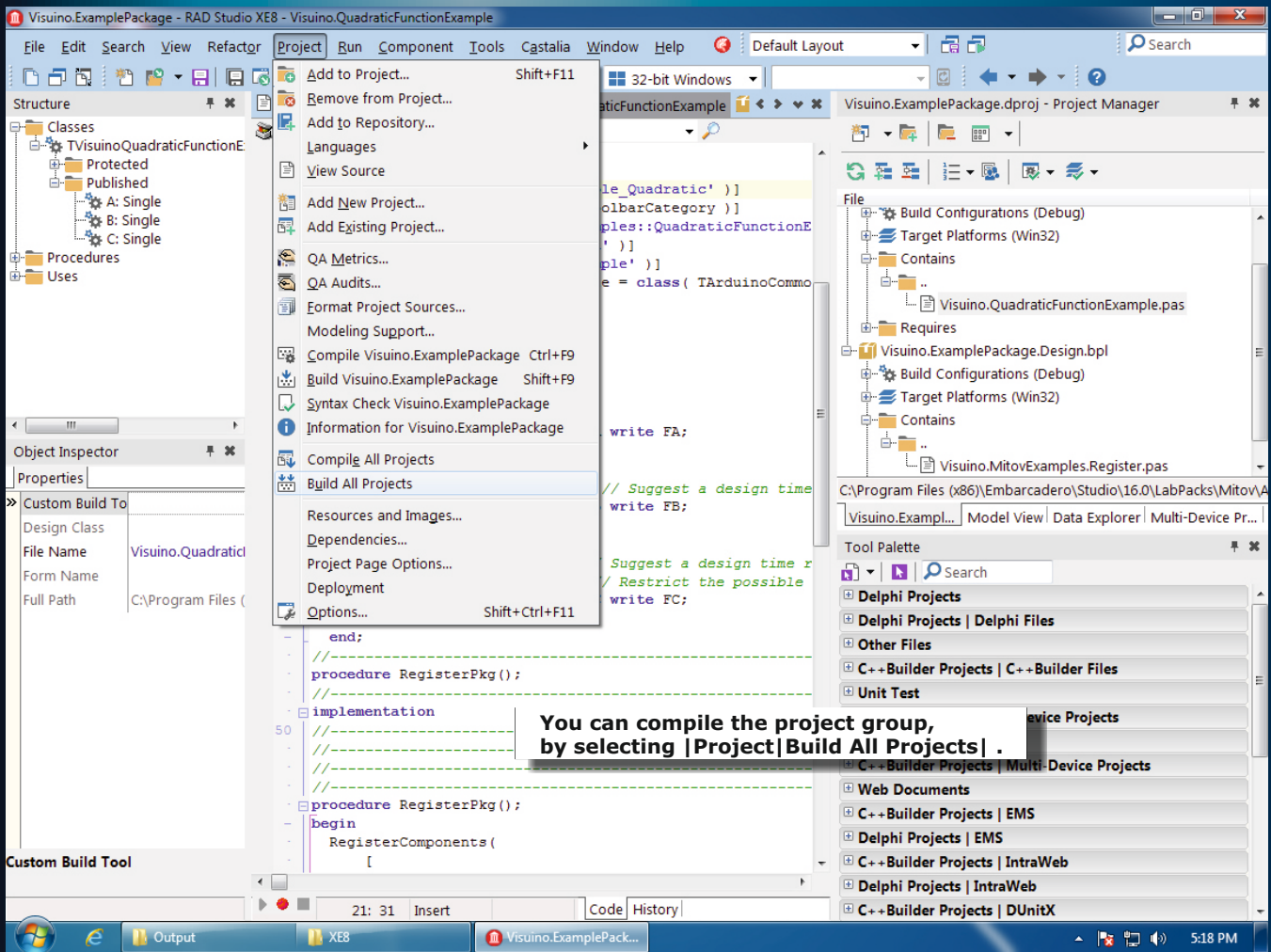
In the `LabPacks\Mitov\Arduino\Examples\XE8` folder, is the `VisuinoExampleProjectGroup.groupproj` project group, containing the projects for the example Visuino package.

If you open the project group, you will discover that it contains 2 projects.

`Visuino.ExamplePackage.dproj`
and
`Visuino.ExamplePackage.Design.dproj`.

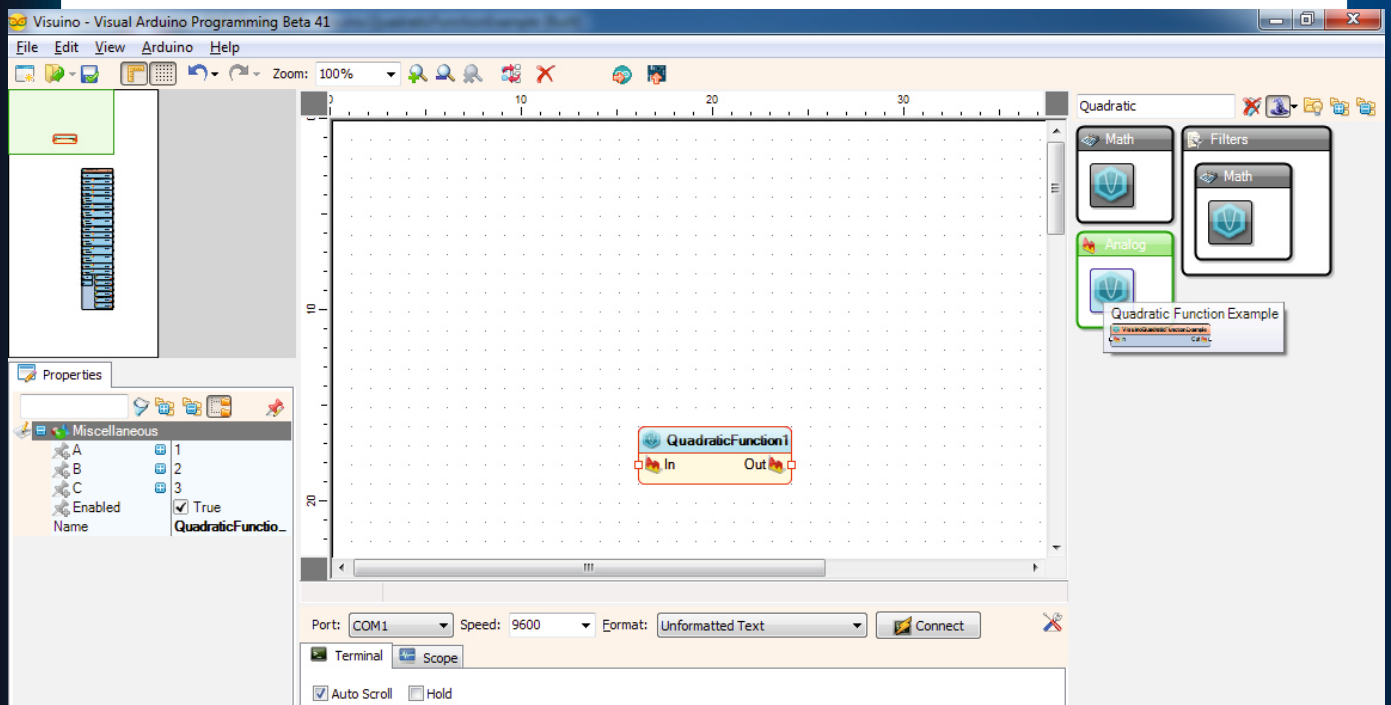
`Visuino.ExamplePackage.dproj` contains the demo components, and the `Visuino.ExamplePackage.Design.dproj` contains the component registration code.

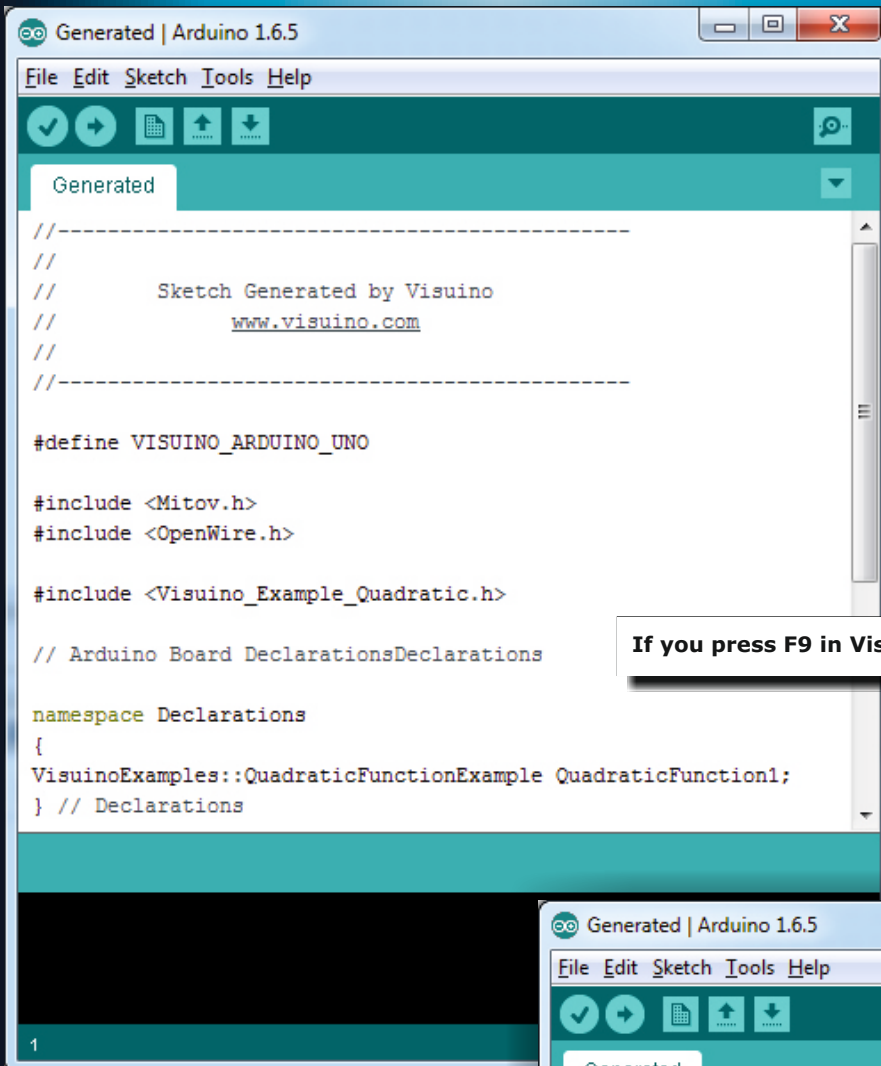




After compiling the project group, in the "LabPacks\Mitov\Arduino\Examples\XE8\Win32\Debug" will contain the compiled packages. To install the example components package, you can copy the *.bpl files into the "Component Packages" directory of Visuino.

If you run Visuino, you will see that a new "Quadratic Function Example" component is available:





```

Generated | Arduino 1.6.5
File Edit Sketch Tools Help
Generated
//-----
//
//      Sketch Generated by Visuino
//      www.visuino.com
//
//-----

#define VISUINO_ARDUINO_UNO

#include <Mitov.h>
#include <OpenWire.h>

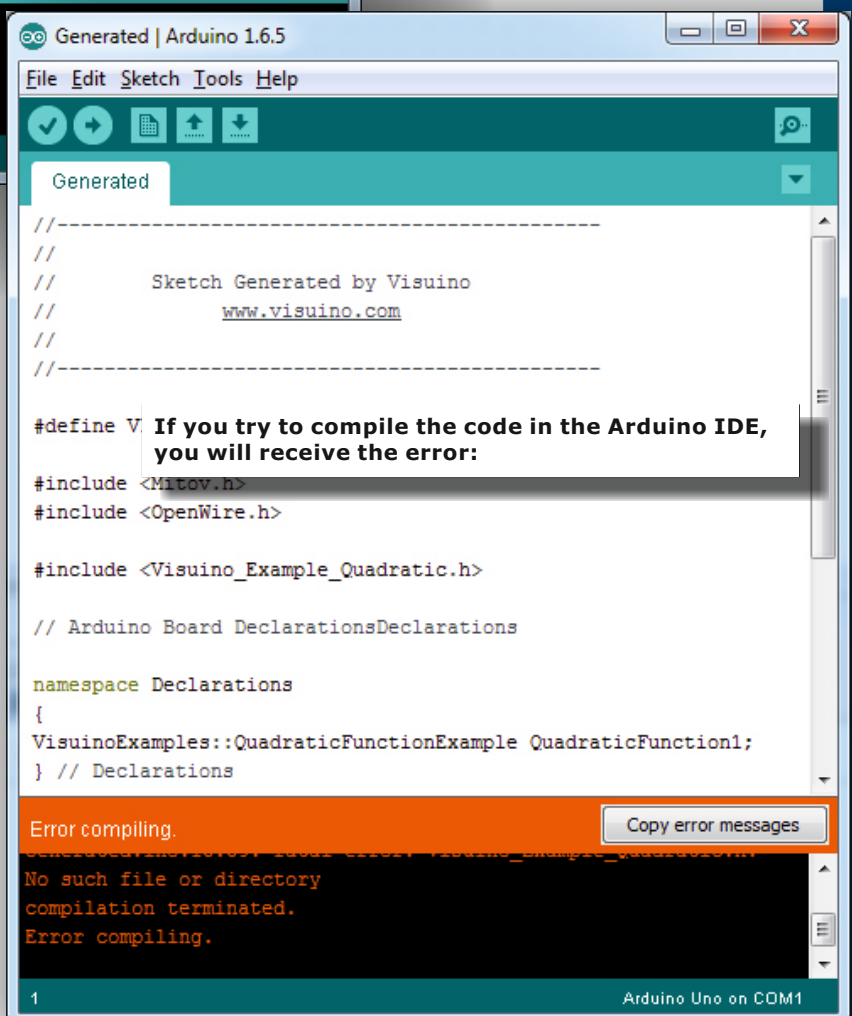
#include <Visuino_Example_Quadratic.h>

// Arduino Board DeclarationsDeclarations

namespace Declarations
{
VisuinoExamples::QuadraticFunctionExample QuadraticFunction1;
} // Declarations
    
```

If you press F9 in Visuino, it will generate the Arduino code.

The reason is that the Arduino IDE can't find the "Visuino_Example_Quadratic.h" file. To make it available to the IDE, you need to copy the file from "LabPacks\Mitov\Arduino\Examples\library" to a sub-directory of "MyDocuments\Arduino\libraries". You can create a directory called "VisuinoExample" - "My Documents\Arduino\libraries\VisuinoExample" and copy the file there. To make the Arduino IDE discover the directory, you will need to close and restart it. The easiest way is by again pressing the **F9** in Visuino after closing the **Arduino IDE**.



```

Generated | Arduino 1.6.5
File Edit Sketch Tools Help
Generated
//-----
//
//      Sketch Generated by Visuino
//      www.visuino.com
//
//-----

#define V
#include <Mitov.h>
#include <OpenWire.h>

#include <Visuino_Example_Quadratic.h>

// Arduino Board DeclarationsDeclarations

namespace Declarations
{
VisuinoExamples::QuadraticFunctionExample QuadraticFunction1;
} // Declarations
    
```

If you try to compile the code in the Arduino IDE, you will receive the error:

Error compiling.

No such file or directory
 compilation terminated.
 Error compiling.

Copy error messages

Arduino Uno on COM1

THIS TIME IF YOU TRY TO COMPILE THE CODE, THERE WILL BE NO ERRORS:

```
Generated | Arduino 1.6.5
File Edit Sketch Tools Help
Generated
//-----
//
//      Sketch Generated by Visuino
//      www.visuino.com
//-----

#define VISUINO_ARDUINO_UNO

#include <Mitov.h>
#include <OpenWire.h>

#include <Visuino_Example_Quadratic.h>

// Arduino Board DeclarationsDeclarations

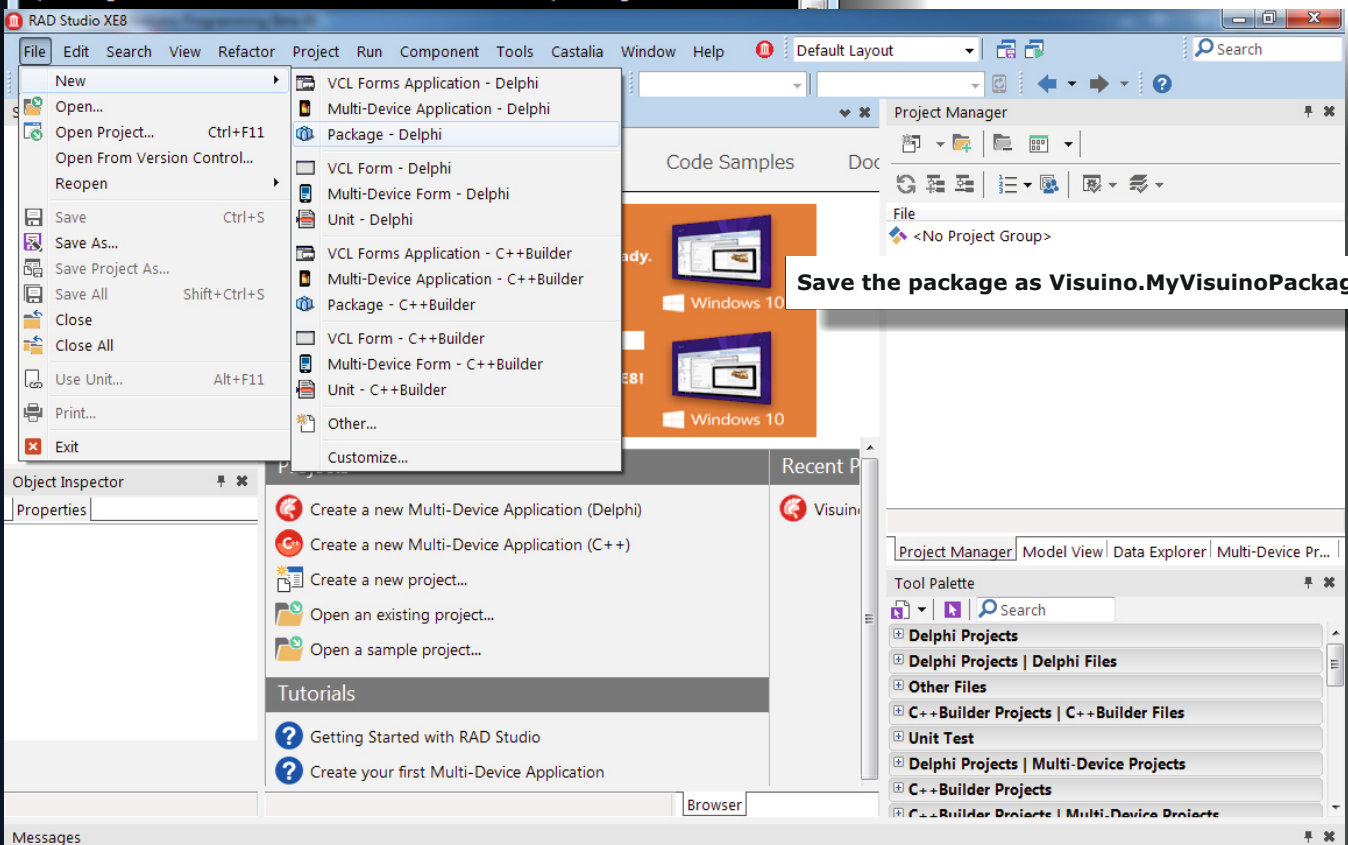
namespace Declarations
{
VisuinoExamples::QuadraticFunctionExample QuadraticFunction1;
} // Declarations

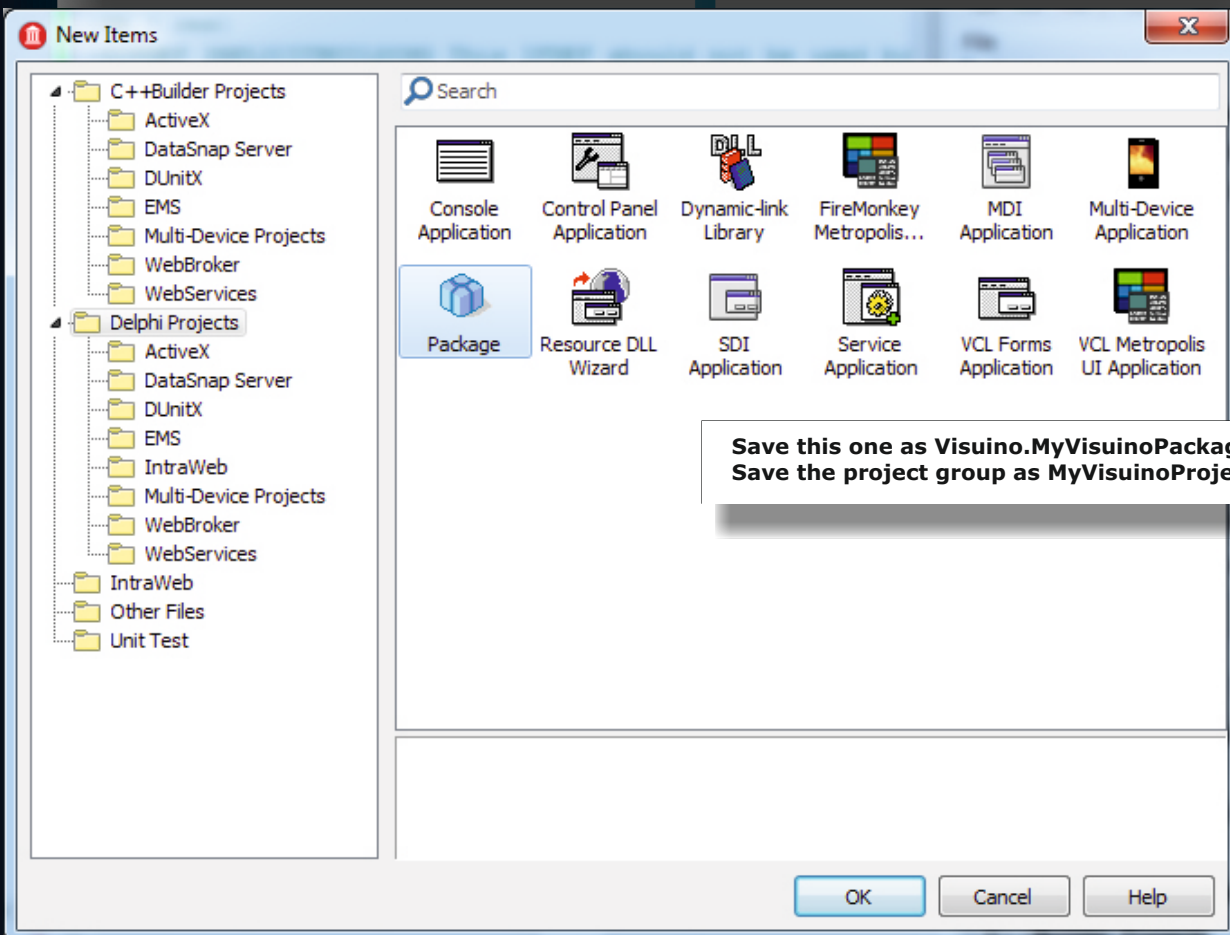
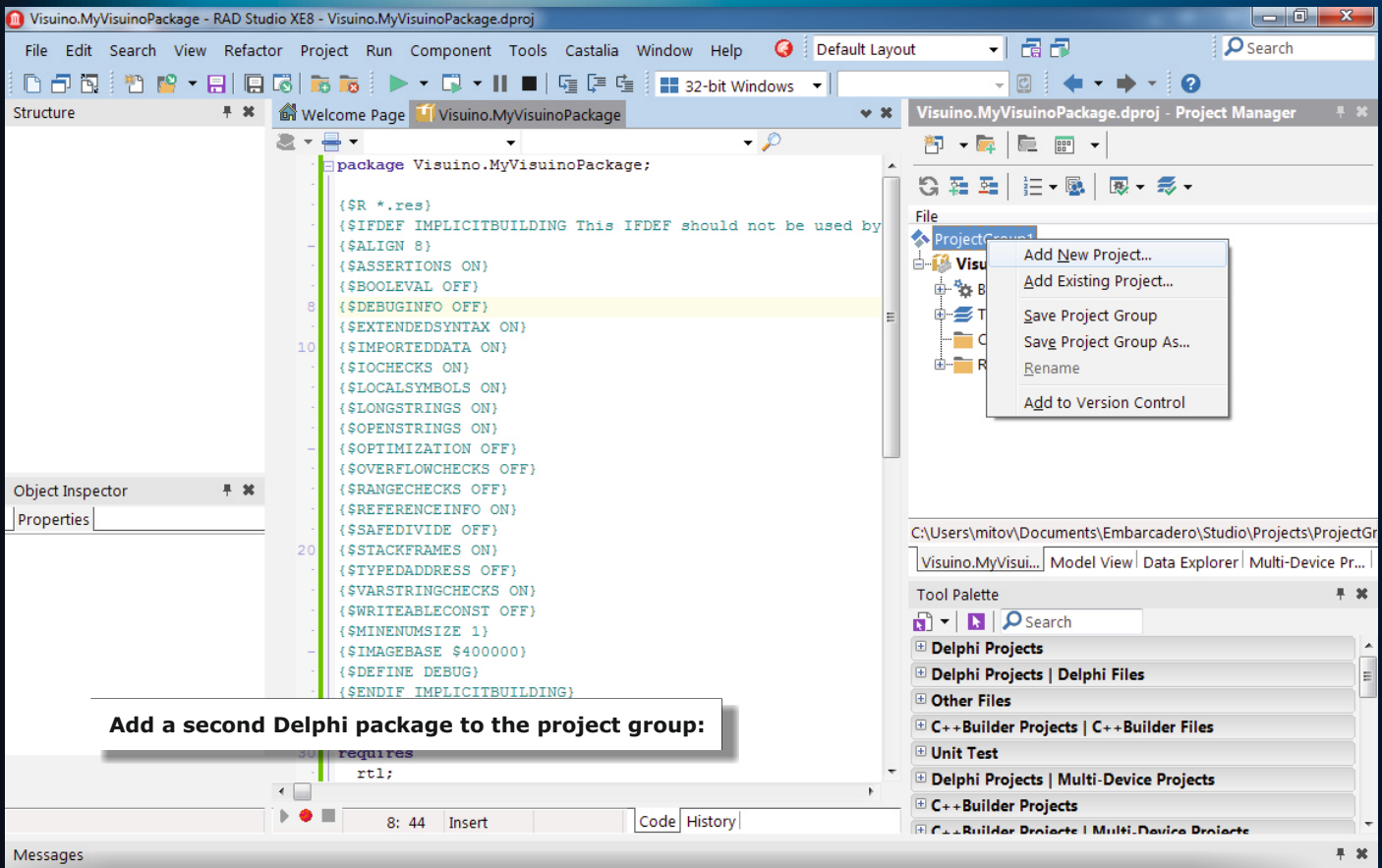
Done compiling.

Global variables use 188 bytes (9%) of dynamic memory, leaving
1,860 bytes for local variables. Maximum is 2,048 bytes.
```

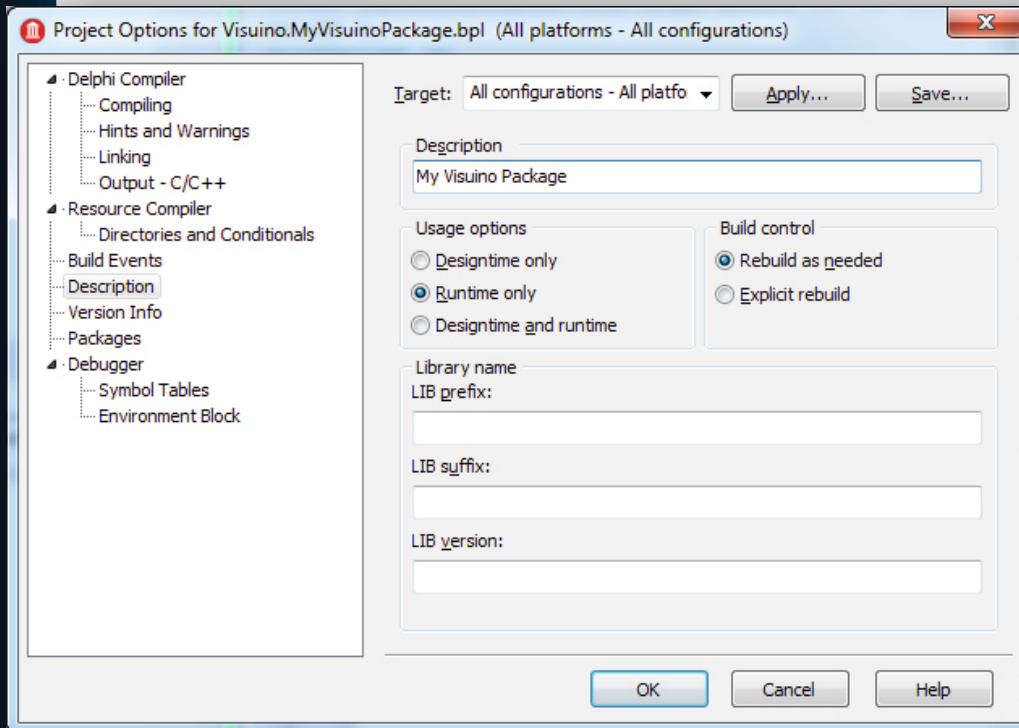
Now after you have learned how to compile and deploy the example packages and components from the Visuino components SDK, it is time to learn how to create packages and components from scratch.

Start by creating a new Delphi package:





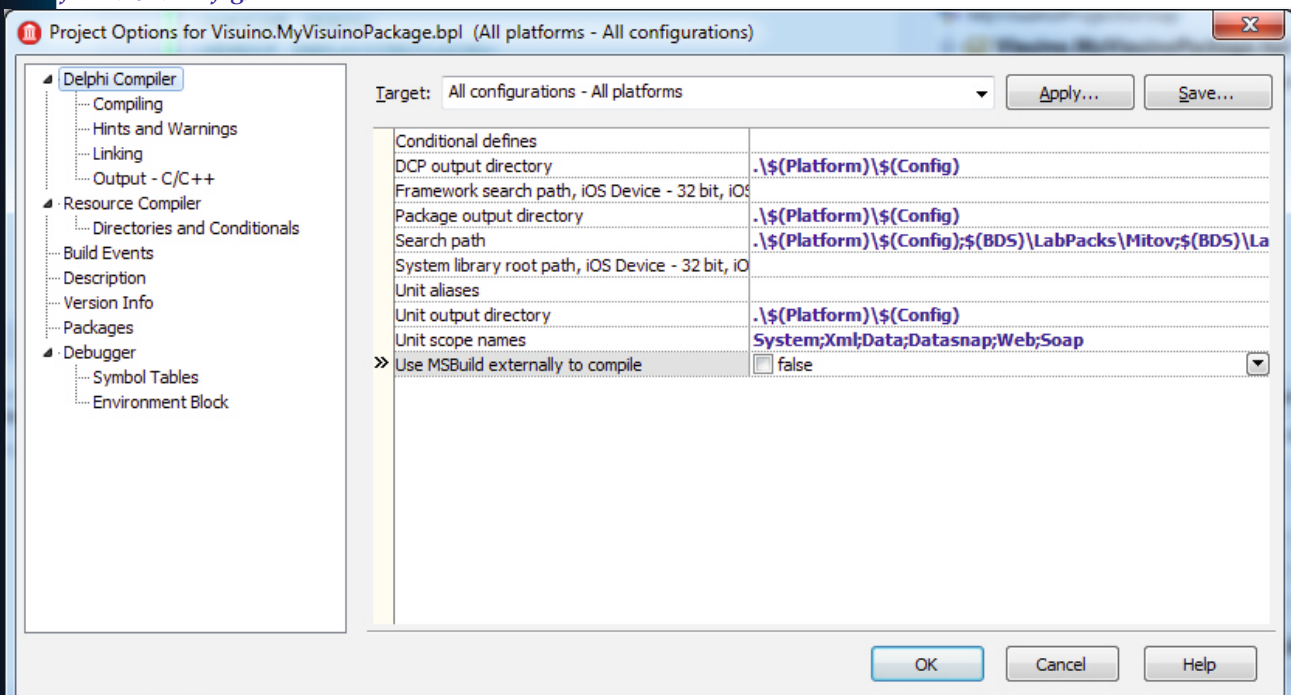
Open the project options for the Visuino.MyVisuinoPackage, select as Target “All configurations”. In the Description settings page, type a description – as example “My Visuino Package”, and set the package to be “Runtime only”:

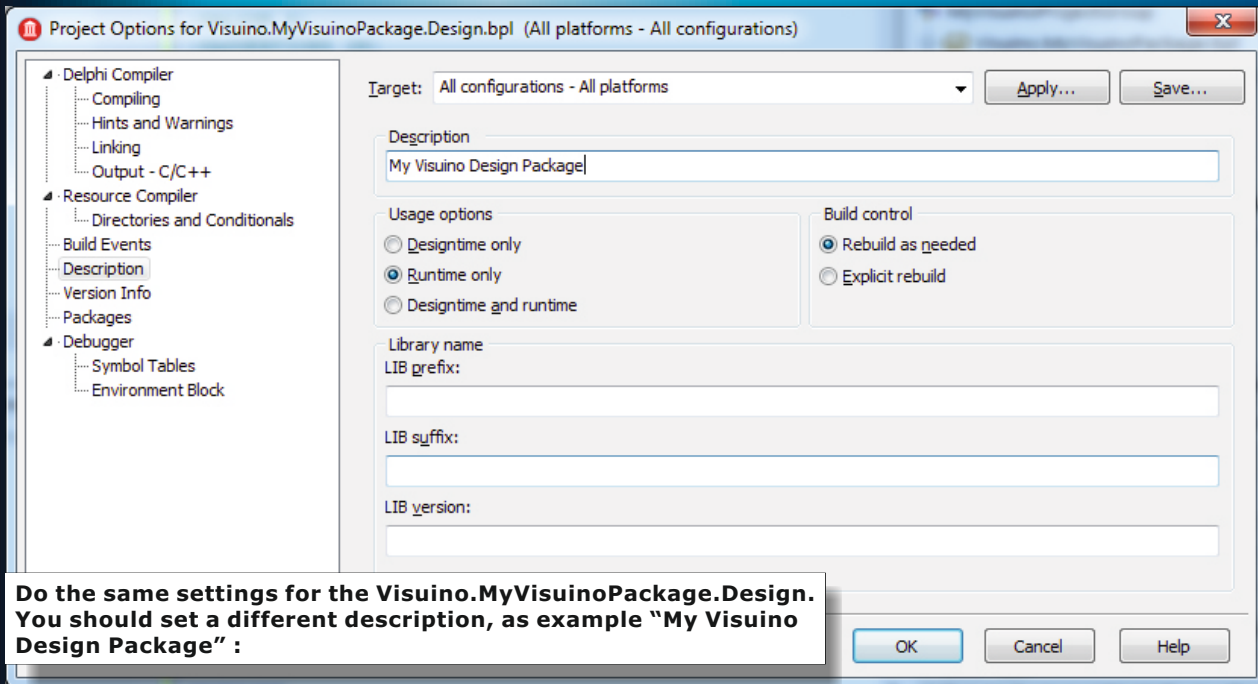


In the “Delphi Compiler” page set: “DCP output directory” and “Package output directory” to: `.\$(Platform)\$(Config)`

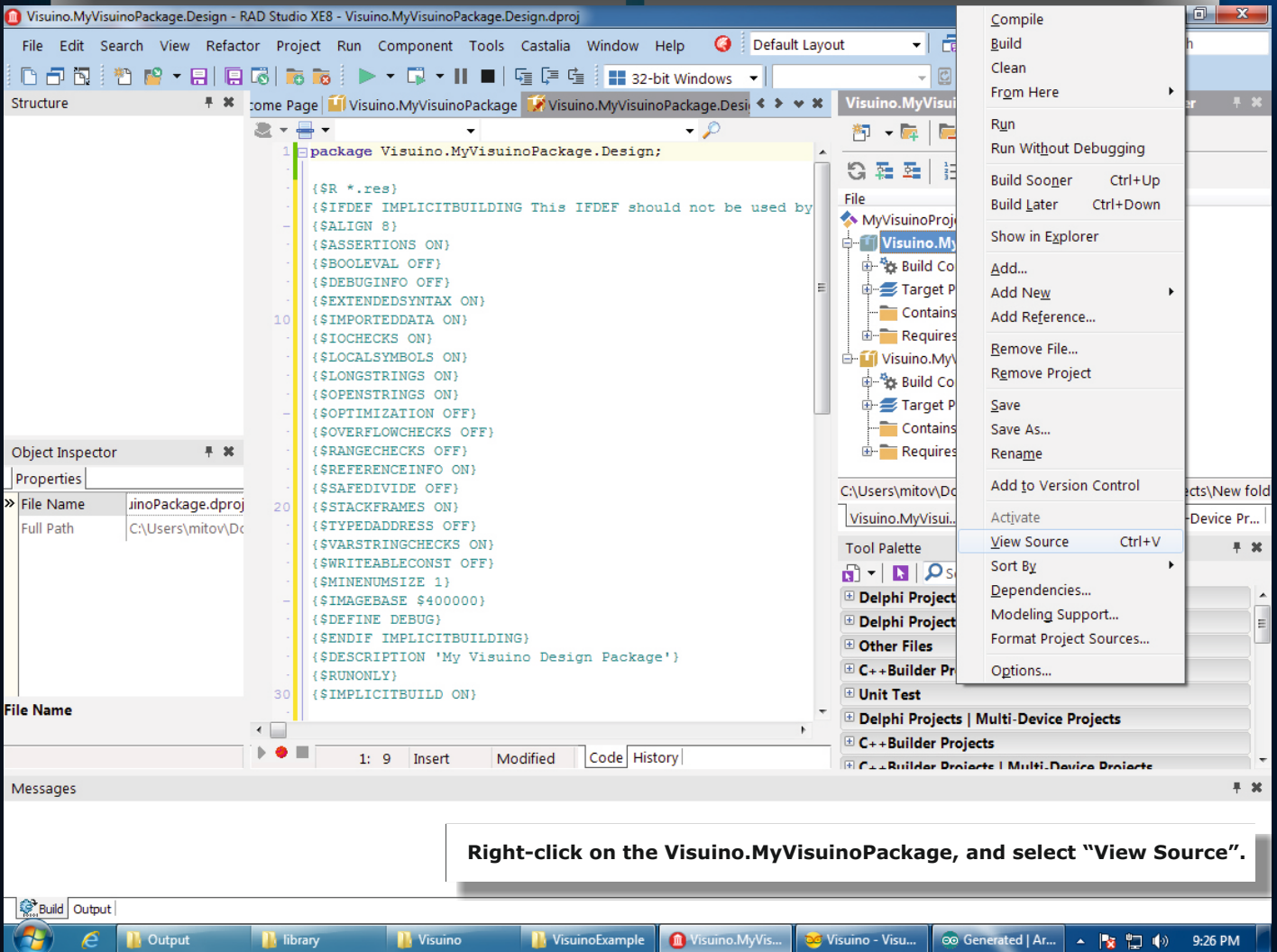
And the “Search Path” to:

`.\$(Platform)\$(Config);$(BDS)\LabPacks\Mitov;$(BDS)\LabPacks\Mitov\Arduino;$(BDS)\LabPacks\Mitov\XE8\$(Platform)\$(Config);$(BDS)\LabPacks\Mitov\Arduino\XE8\$(Platform)\$(Config)`

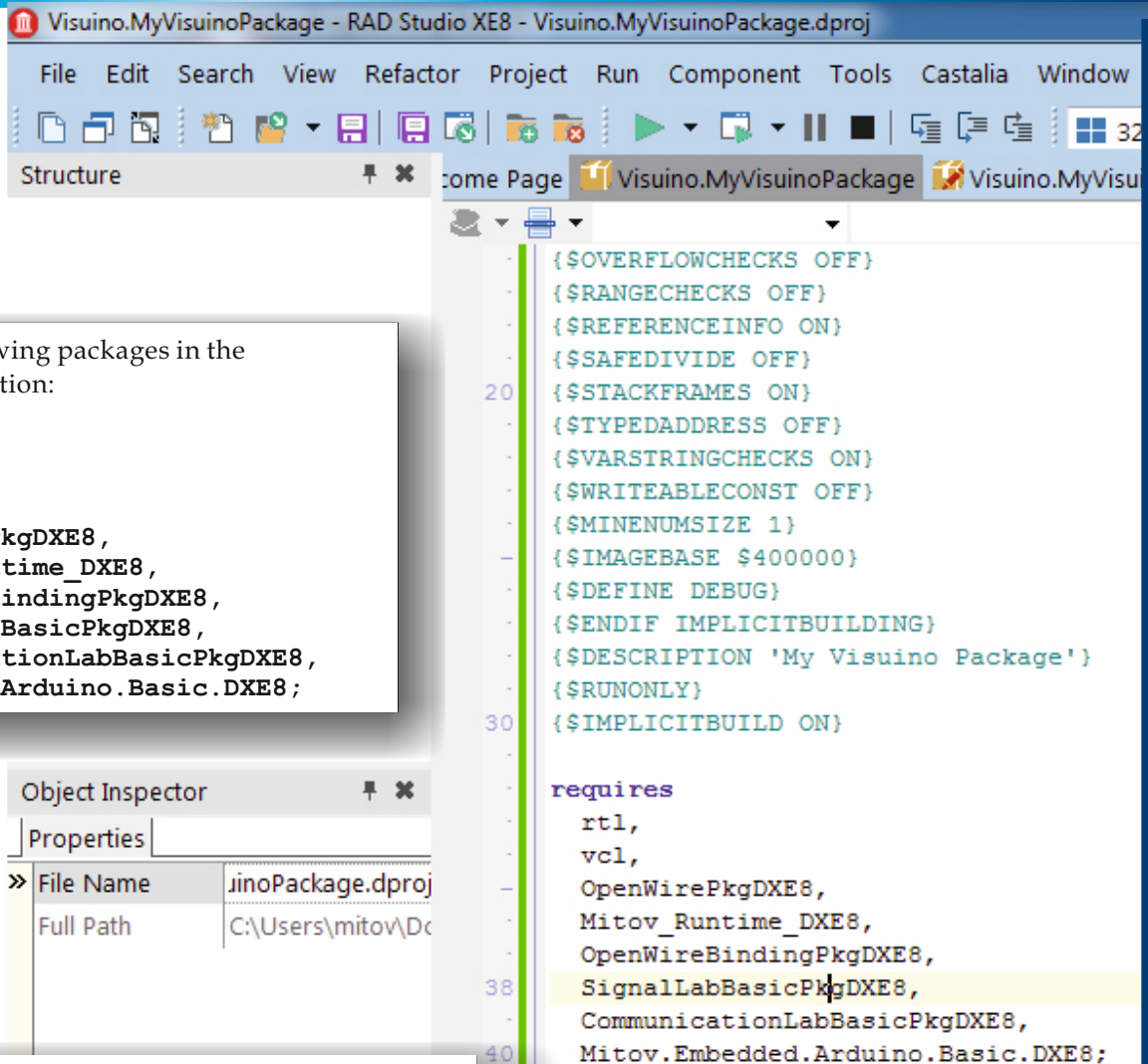




Do the same settings for the Visuino.MyVisuinoPackage.Design. You should set a different description, as example "My Visuino Design Package" :



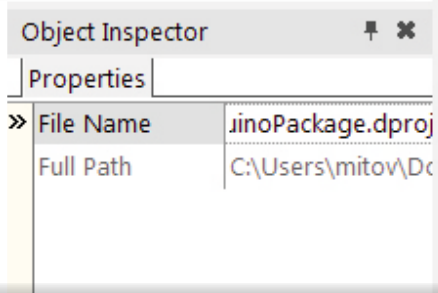
Right-click on the Visuino.MyVisuinoPackage, and select "View Source".



Add the following packages in the "requires" section:

```

requires
    rtl,
    vcl,
    OpenWirePkgDXE8,
    Mitov_Runtime_DXE8,
    OpenWireBindingPkgDXE8,
    SignalLabBasicPkgDXE8,
    CommunicationLabBasicPkgDXE8,
    Embedded.Arduino.Basic.DXE8;
    
```



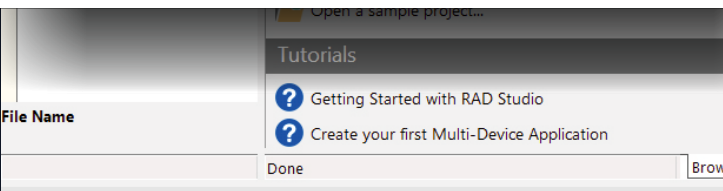
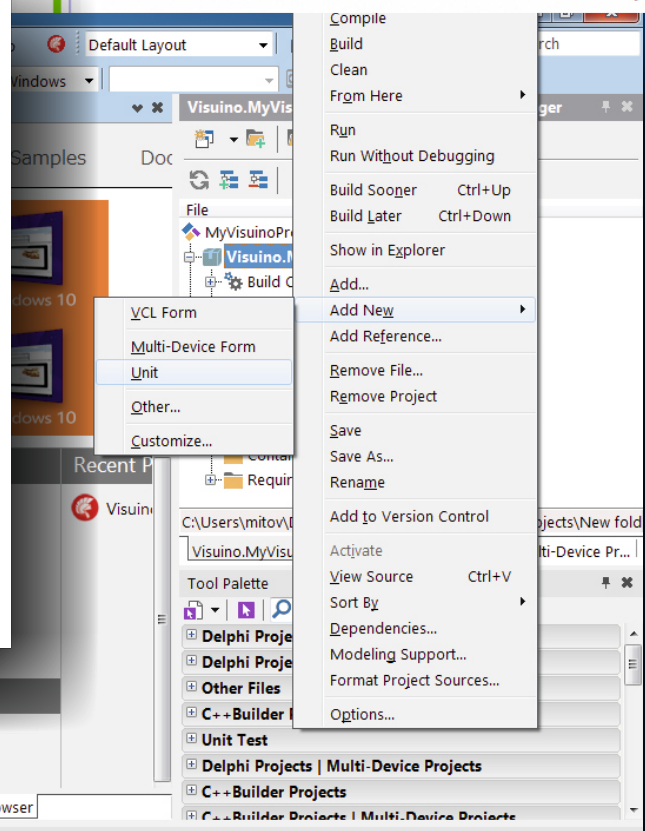
Do the same for the Visuino.MyVisuinoPackage.Design, however this time add to the "requires" section:

```

requires
    rtl,
    vcl,
    OpenWirePkgDXE8,
    Mitov_Runtime_DXE8,
    OpenWireBindingPkgDXE8,
    SignalLabBasicPkgDXE8,
    CommunicationLabBasicPkgDXE8,
    Mitov.Embedded.Arduino.Basic.DXE8,
    Visuino.MyVisuinoPackage;
    
```

Now that you have your packages ready, you can start working on your component. You will add component almost identical to the one in the included example, so you can also copy and paste some of the code from there.

Add new unit to the **Visuino.MyVisuinoPackage**:



Save the new unit as Visuino.MyComponent.
 In the unit add:

```
unit Visuino.MyComponent;

interface

uses
    System.Classes, Mitov.Design.Components,
    Mitov.Arduino.Types;

type
    TVisuinoMyComponent =
        class( TArduinoCommonAnalogFilter )
        end;

procedure RegisterPkg();

implementation

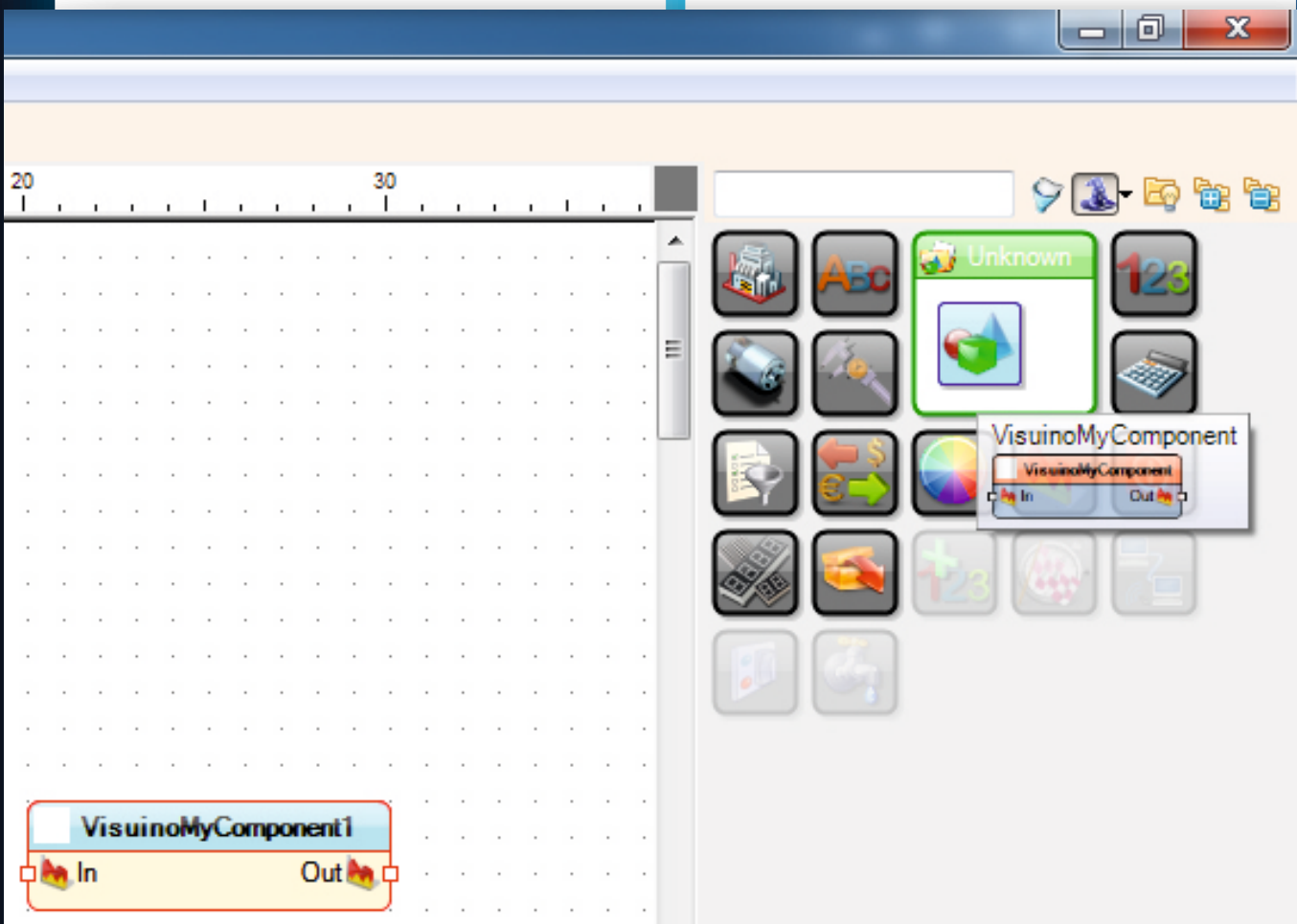
procedure RegisterPkg();
begin
    RegisterComponents(
        [
            TVisuinoMyComponent
        ]
    );
end;

end.
```

Add a unit to the **Visuino.MyVisuinoPackage.Design**, and save it as **Visuino.MyPackage.Register**. In this unit add:

```
unit Visuino.MyPackage.Register;
interface
procedure Register();
implementation
uses Visuino.MyComponent;
procedure Register();
begin
    Visuino.MyComponent.RegisterPkg();
end;
end.
```

When Visuino loads the **Visuino.MyVisuinoPackage.Design** package it will call the Register() method. The Register method will call **Visuino.MyComponent.RegisterPkg()**; **RegisterPkg()** will call the **RegisterComponents**, and will pass array of components to be registered in the Visuino toolbar. If you install the packages in Visuino and run it. You will see that our new component is available:



The component however is not registered in any categories, does not have image, and does not have user friendly name. Lets start fixing this. Add `Mitov.Arduino.Categories.Basic` to the uses clause, and `[Category(TArduinoMathFilterToolBarCategory)]` attribute to the component:

```
uses
    System.Classes, Mitov.Design.Components,
    Mitov.Arduino.Types,
    Mitov.Arduino.Categories.Basic;

type
    [Category( TArduinoMathFilterToolBarCategory )]
    TVisuinoMyComponent =
        class( TArduinoCommonAnalogFilter )
        end;
```

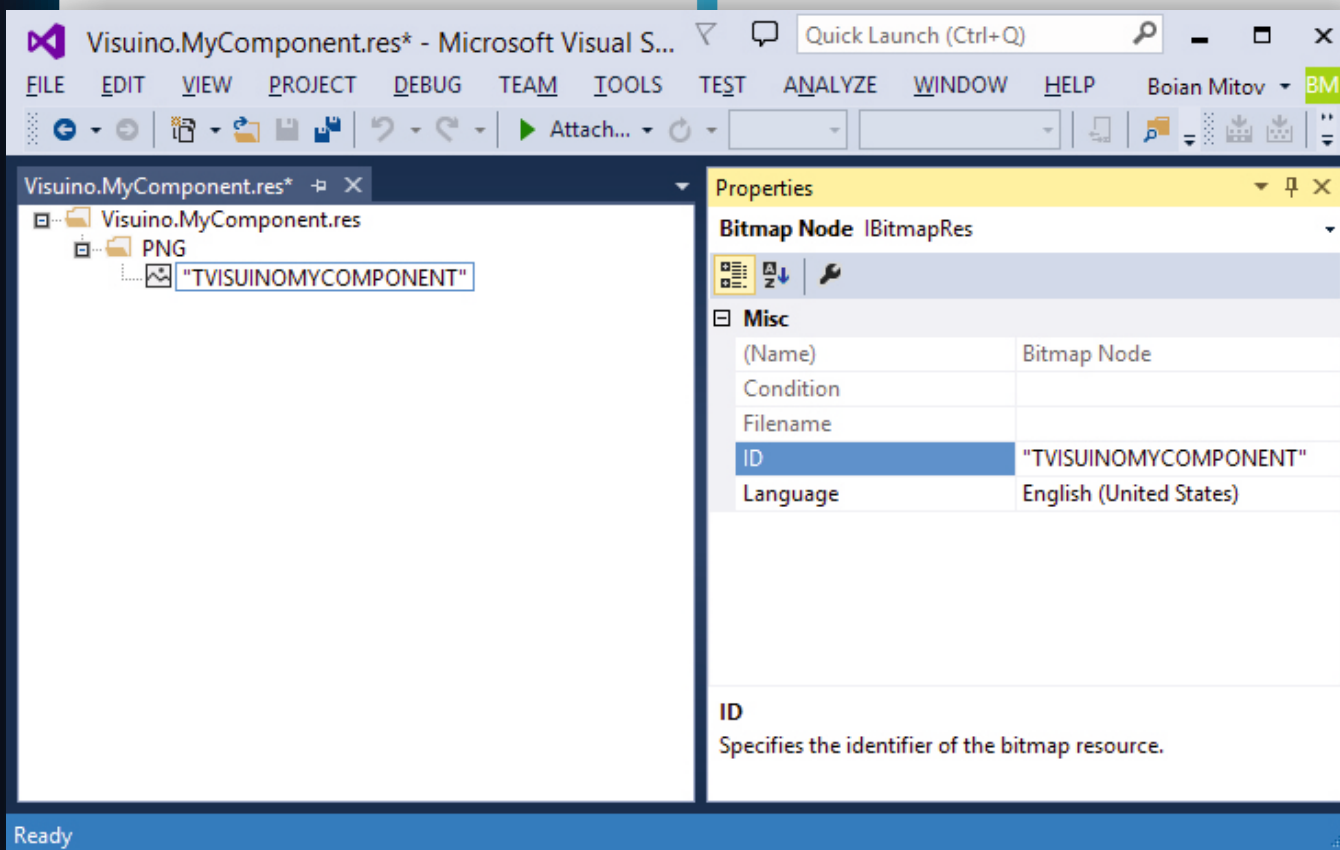
This will place the component in the Arduino/Filters/Math subcategory, and the associated alternative categories.

To add the desired user friendly names, add `Mitov.Attributes` to the uses, and `[CreateName('MyVisuinoComponent')]`, and `[Name('My First Visuino Component')]` attributes to the component:

```
System.Classes, Mitov.Design.Components,
Mitov.Arduino.Types,
Mitov.Arduino.Categories.Basic,
Mitov.Attributes;

type
    [Category(TArduinoMathFilterToolBarCategory)]
    [CreateName( 'MyVisuinoComponent' )]
    [Name( 'My First Visuino Component' )]
    TVisuinoMyComponent =
        class( TArduinoCommonAnalogFilter )
        end;
```

You can also add image for the component. To do that, you need to create an empty `Visuino.MyComponent.res` resource file, and add 32x32 pixels PNG image as resource named `"TVISUINOMYCOMPONENT"` - upper case of our component name as declared in the code. I usually use Visual Studio to create and edit the resource files, but you can use any other resource editor.



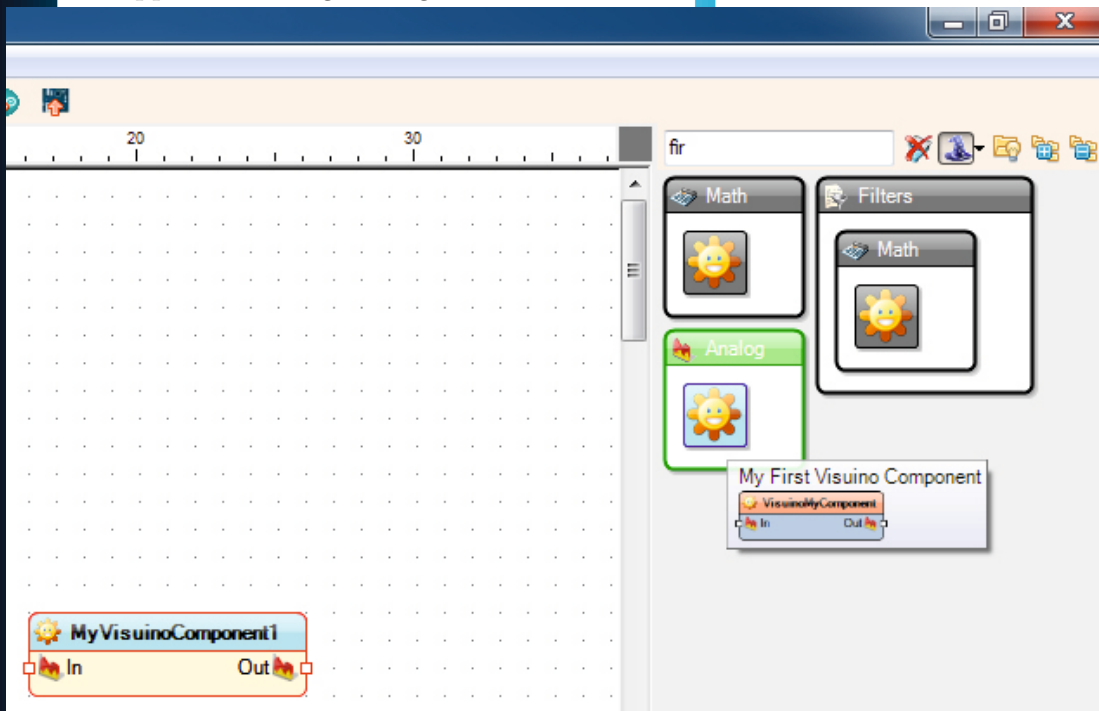
You can also open and explore the "Visuino.QuadraticFunctionExample.res" included in the SDK example. To use the resource in the component, add:

```
unit Visuino.MyComponent;
```

```
{$R *.res}
```

to the Visuino.MyComponent.pas file. Now if you rebuild, and deploy the package and run Visuino, you will see that the component has the new names and image, and appears in the right categories:

This is just the code to have an empty header file that has define preventing it from being included multiple times, and includes the Mitov.h header file that contains the Visuino base classes. Next you will create a namespace where you can put our component. This step is not really required, but is highly recommended to avoid conflicts of multiple components from multiple vendors with the same name. Use your own name, or company name for the namespace, or something else that is likely to be unique.



I mentioned earlier that each Visuino component has 2 parts. The Arduino C++ code, and its visual representation in Visuino. So far you have created a visual representation of such Arduino component. Now lets write the C++ code. Under "Arduino\libraries" create "MyExample" folder: "My documents\Arduino\libraries\MyExample". In this folder create a new .h file Visuino_MyComponent.h, and open the file to edit it as example in RAD Studio, Visual Studio, Notepad or other editor, and write the minimal code necessary:

```
#ifndef VISUINO_MY_COMPONENT_h
#define VISUINO_MY_COMPONENT_h

#include <Mitov.h>

namespace MyVisuinoComponents
{
}

#endif
```

In the namespace now you can declare your component:

```
#ifndef VISUINO_MY_COMPONENT_h
#define VISUINO_MY_COMPONENT_h

#include <Mitov.h>

#endif
```

```
namespace MyVisuinoComponents
{
    class MyComponent : public Mitov::CommonEnableFilter
    {
        typedef Mitov::CommonEnableFilter inherited;
    };
}
```

Now that you have a rudimentary Visuino component that you can test, go back to the Delphi code, and instruct it how to generate code for the component.

First you need to specify the header file that needs to be included in the Arduino project when you use the component. In this case

Visuino_MyComponent.h. You can do this by adding `[ArduinoInclude('Visuino_MyComponent')]` attribute.

Second you need to specify the namespace and the name of the C++ component. You can do this by adding `[ArduinoComponent('MyVisuinoComponents::MyComponent')]` attribute:

```
type
[ArduinoInclude( 'Visuino_MyComponent' )]
[Category( TArduinoMathFilterToolbarCategory )]
[ArduinoComponent( 'MyVisuinoComponents::MyComponent' )]
[CreateName( 'MyVisuinoComponent' )]
[Name( 'My First Visuino Component' )]
TVisuinoMyComponent = class( TArduinoCommonAnalogFilter
)
end;
```

This is enough to generate the proper code for Arduino, however the `MyComponent` inherits from `Mitov::CommonEnableFilter` which contains abstract method `virtual void DoReceive(void *_Data)`.

You need to implement this method in your C++ code, and this is where our data processing will be done. For now you can just do nothing with the data, and send it to the output of the filter without a change:

```
class MyComponent : public Mitov::CommonEnableFilter
{
    typedef Mitov::CommonEnableFilter inherited;

protected:
    virtual void DoReceive( void *_Data )
    {
        OutputPin.Notify( _Data );
    }
}
```

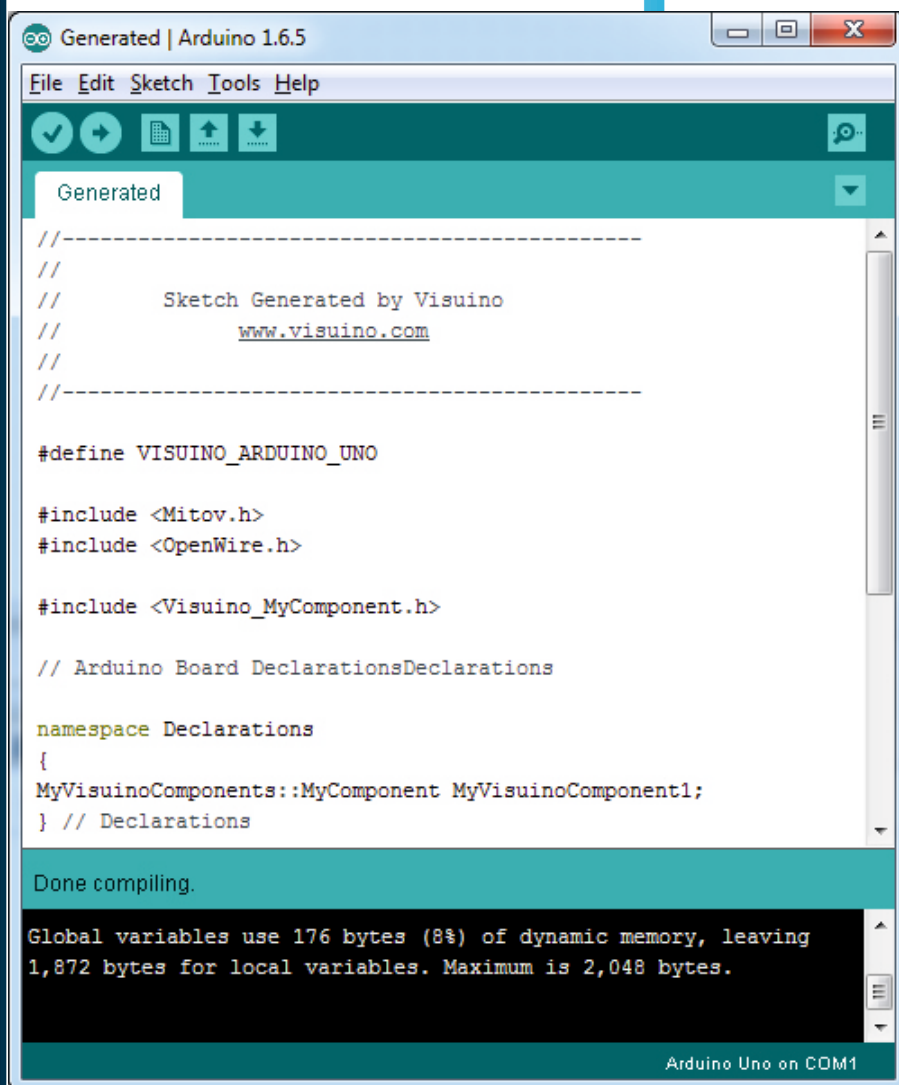
`DoReceive` receives a pointer to the data, and it just calls `OutputPin.Notify` passing the same pointer to the data to be sent to the next filter in the chain. If you generate

the code, and compile it in the Arduino IDE, it will succeed: see figure at left.

Your first component is done, but it does not do much.

To be useful it needs to perform some processing over the data.

Now you will implement the same quadratic function as implemented in the SDK's example, but you can easily implement on your own any other function you can think of.



For the quadratic function you will need 3 properties containing the 3 coefficients A, B and C. So lets declare them in the C++ code first:

```
class MyComponent : public Mitov::CommonEnableFilter
{
    typedef Mitov::CommonEnableFilter inherited;
public:
    float A,B,C;
```

Next, you need to initialize them with their default values in the constructor:

```
class MyComponent : public Mitov::CommonEnableFilter
{
    typedef Mitov::CommonEnableFilter inherited;

public:
    float A,B,C;

protected:
    virtual void DoReceive( void *_Data )
    {
        OutputPin.Notify( _Data );
    }

public:
    MyComponent () :
        A( 1.0 ), B( 2.0 ), C( 3.0 )
    {
    }
};
```

Finally you need to implement the proper computation in the DoReceive:

```
protected:
virtual void DoReceive( void *_Data )
{
    if( Enabled )
    {
        float AValue = *(float *)_Data;
        AValue = AValue * AValue * A + AValue * B + C;
        OutputPin.Notify( &AValue );
    }

    else
        OutputPin.Notify( _Data );
```

In this case if the component is Enabled it will obtain the floating point value from the _Data pointer, use it for the calculation, and then send the result trough the OutputPin . If the component is not Enabled it will just send the data trough the OutputPin without changes. The C++ code for the component is ready. Now lets add the properties in the Delphi code.

First you need to declare the 3 fields that will hold the properties:

```
TVisuinoMyComponent = class ( TArduinoCommonAnalogFilter )
protected
    FA : Single; FB : Single; FC : Single;
end;
```

Next you need to declare the 3 properties, and specify the default values. The Visuino component framework will initialize the fields automatically with the default values:

```
TVisuinoMyComponent = class( TArduinoCommonAnalogFilter )
protected
    FA : Single;    FB : Single;    FC : Single;

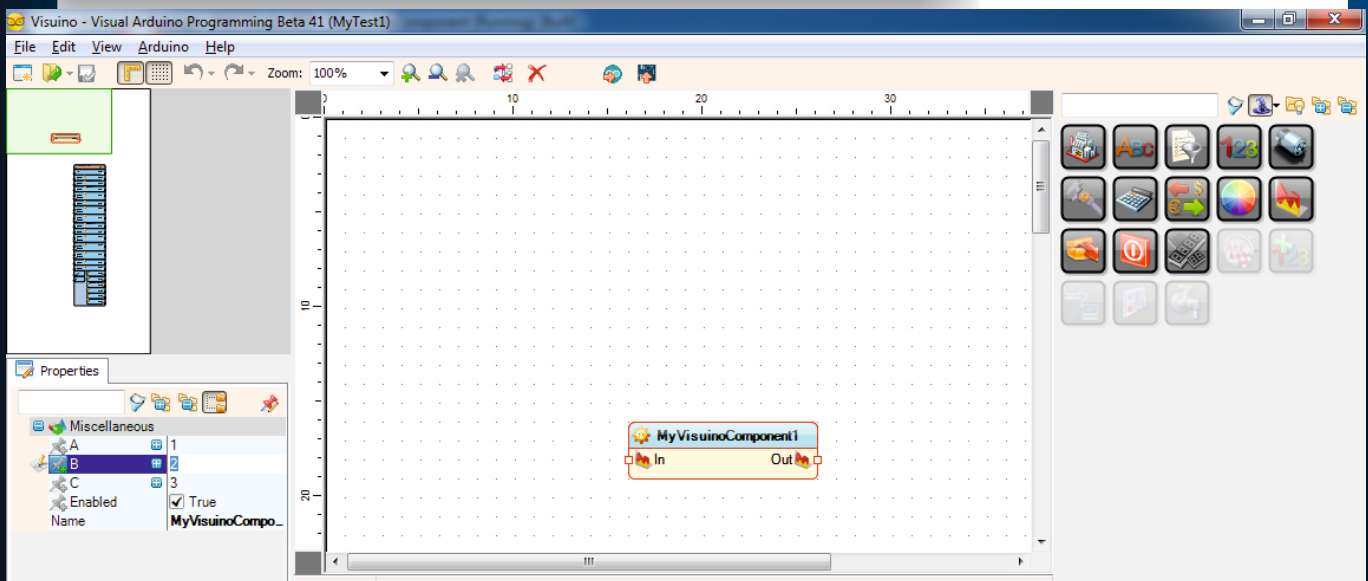
published
    [DefaultSingle( 1.0 )]
    property A : Single read FA write FA;

    [DefaultSingle( 2.0 )]
    property B : Single read FB write FB;

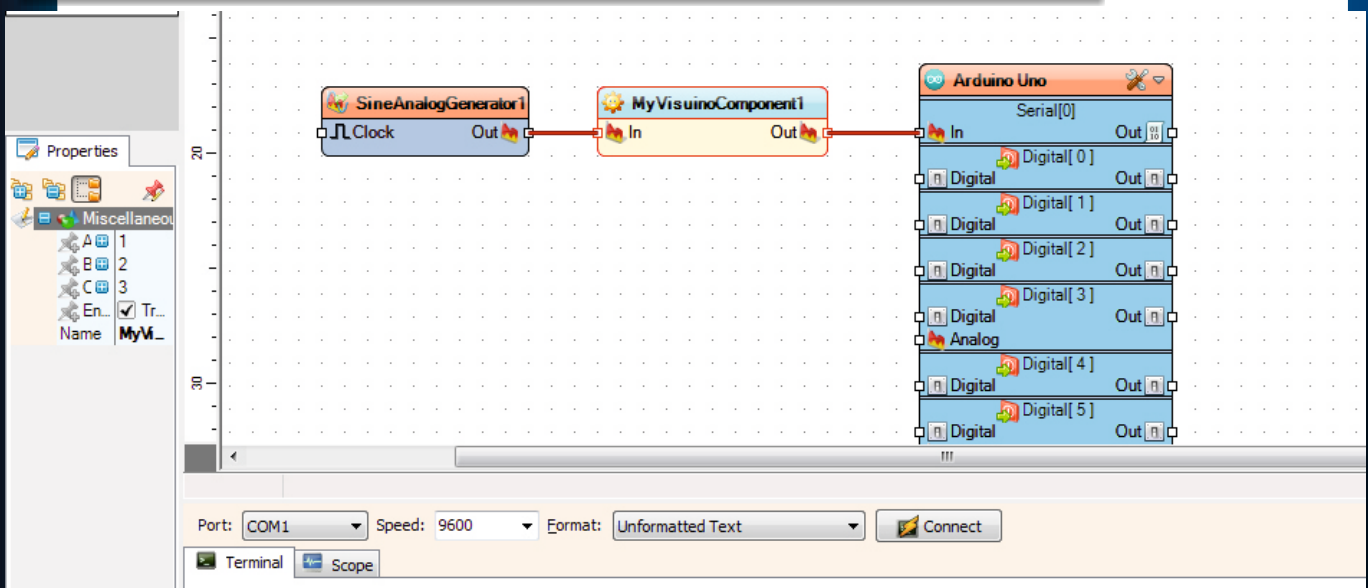
    [DefaultSingle( 3.0 )]
    property C : Single read FC write FC;

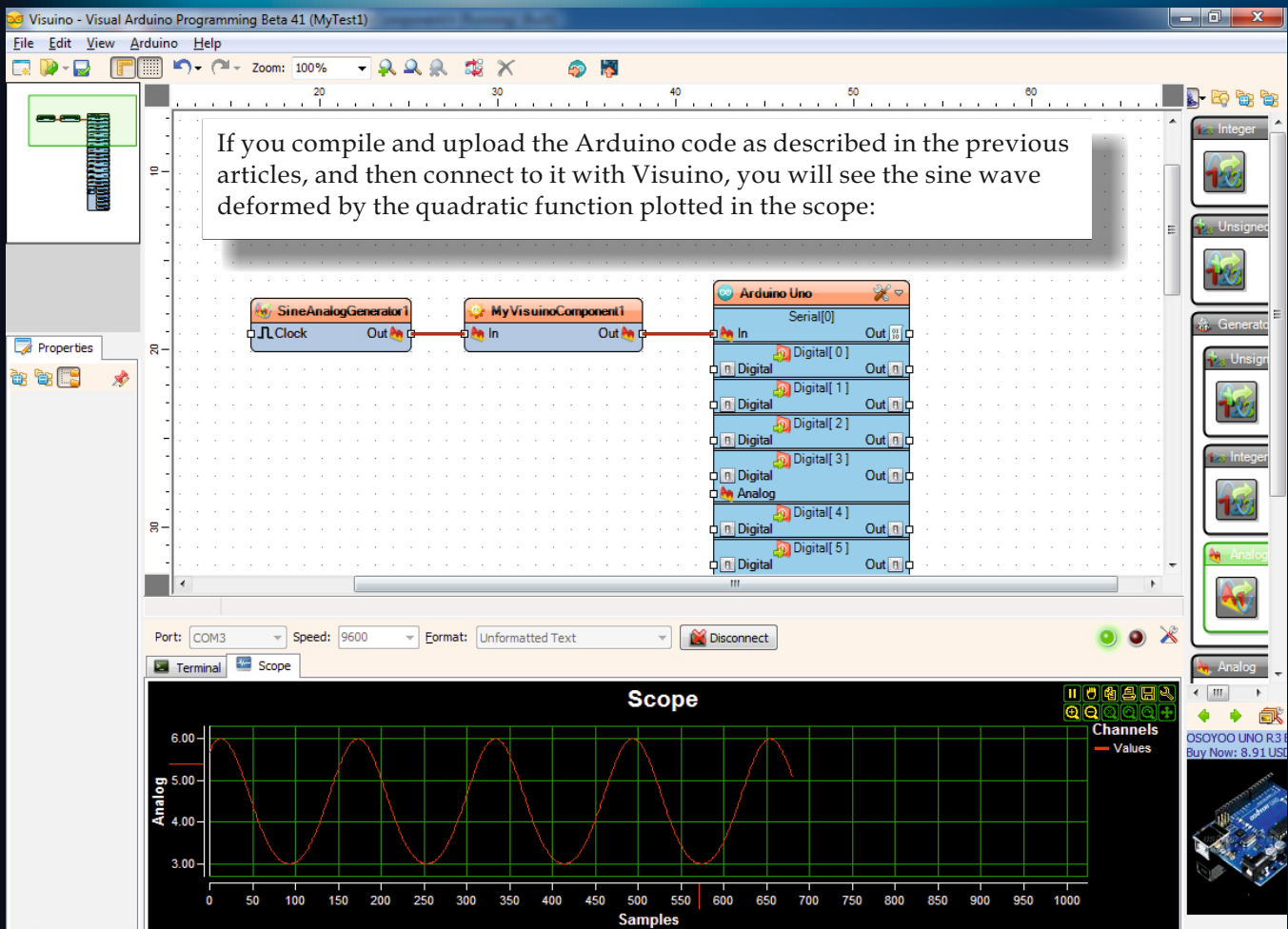
end;
```

Your component is ready to be used. If you deploy the compiled *.bpl files to the “Component Packages” sub-directory of Visuino, and run Visuino, you will have the properties available, and you can edit them:



Here is example Visuino connection diagram that you can use to test the new component:





Your component is ready, tested and operational.

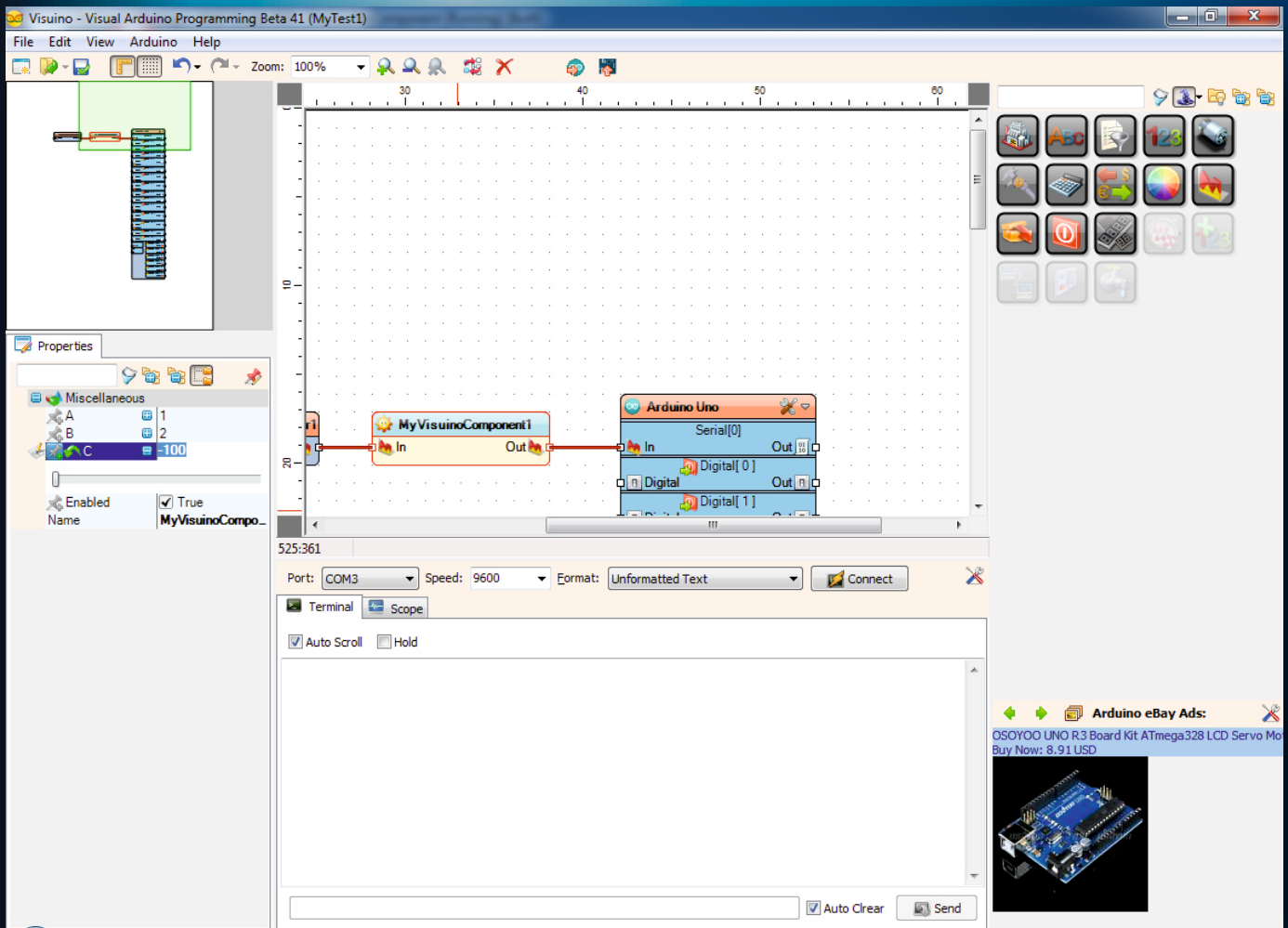
There are some final improvements that you can do. At the moment the end user of the component can set any value for the A, B and C parameters. Sometimes you want to limit that range. You can limit the range of a property by adding the `ValueRange` attribute, as example like this:

```
[DefaultSingle( 3.0 )]
[ValueRange( -1000, 1000 )]
property C : Single read FC write FC;
```

You can also add a suggested smaller range that will be used in the Visuino property editor when showing the in-place track-bar value editor, by using the `DesignRange` attribute like this:

```
[DefaultSingle( 3.0 )]
[DesignRange( -100, 100 )]
[ValueRange( -1000, 1000 )]
property C : Single read FC write FC;
```

If you add those attributes, rebuild and deploy the package, in Visuino the in-place editor will offer only the -100 to 100 range:



Any attempt to also enter manually value outside the -1000 to 1000 range will fail. If the DesignRange attribute is not present, and ValueRange is present, the track-bar will use the ValueRange instead. If none of them is present, the track-bar will offer the full range of floating point values.

CONCLUSION

In this article you learned the basics of creating your own Visuino components. As you have seen, it is very easy, and can be done by almost anyone, even with limited programming knowledge.

The component we demonstrated is very simple, and yet already very useful. With the component SDK you can create much more complex and advanced components, with many more features. All components included in Visuino are written using this SDK, and you can see the power they offer. It is not possible in a single article to cover all aspects of Visuino component development, but this is a very good starting point.

We are working hard to provide more resources and information on Visuino component development, and I hope you all will have many joyful hours playing with Visuino and creating your own components for it.

In the next Visuino article we will show you how you can connect Delphi applications and Arduino boards over internet, and how you can make different Arduino boards talk to each other. You will be entering the exciting world of "Internet of Things"!

